Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security

Mohit Tiwari*, Jason K Oberg†, Xun Li*, Jonathan Valamehr*, Timothy Levin‡
Ben Hardekopf*, Ryan Kastner†, Frederic T. Chong*, Timothy Sherwood*

* Dept. of Computer Science, University of California, Santa Barbara, CA, USA
† Dept. of Computer Science and Engineering, University of California, San Diego, CA, USA
‡ Naval Postgraduate School, Monterey, CA, USA

{tiwari, xun, valamehr, benh, chong, sherwood}@cs.ucsb.edu
{jkoberg, kastner}@cs.ucsd.edu, levin@nps.edu

ABSTRACT

High assurance systems used in avionics, medical implants, and cryptographic devices often rely on a small trusted base of hardware and software to manage the rest of the system. Crafting the core of such a system in a way that achieves flexibility, security, and performance requires a careful balancing act. Simple static primitives with hard partitions of space and time are easier to analyze formally, but strict approaches to the problem at the hardware level have been extremely restrictive, failing to allow even the simplest of dynamic behaviors to be expressed.

Our approach to this problem is to construct a minimal but configurable *architectural skeleton*. This skeleton couples a critical slice of the low level hardware implementation with a microkernel in a way that allows information flow properties of the entire construction to be statically verified all the way down to its gate-level implementation. This strict structure is then made usable by a runtime system that delivers more traditional services (e.g. communication interfaces and long-living contexts) in a way that is decoupled from the information flow properties of the skeleton. To test the viability of this approach we design, test, and statically verify the information-flow security of a hardware/software system complete with support for unbounded operation, inter-process communication, pipelined operation, and I/O with traditional devices. The resulting system is provably sound even when adversaries are allowed to execute arbitrary code on the machine, yet is flexible enough to allow caching, pipelining, and other common case optimizations.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/ software interfaces*; C.3 [**Special Purpose And Application Based Systems**]: [Real-time and Embedded Systems]

General Terms

Design, Security, Reliability, Verification

Keywords

High Assurance Systems, Gate Level Information Flow Tracking, Non-interference

ISCA'11, June 4–8, 2011, San Jose, California, USA.

1. INTRODUCTION

Systems requiring the highest levels of trust are often designed and built using waterfall principles, modeled at a high-level in a theorem proving system, coaxed through said theorem proving system by hand, documented to an tremendous degree throughout the entire process, and then evaluated by a trusted third party or evaluation board [2]. It is estimated that the entire process costs over $10,000 per line of code [4] and takes over 10 years to complete [3]. In the end, there is more evaluation of the *development process* than the final *end artifact*, and formal properties are shown to hold only for hand-written high-level models of the system rather than for the actual implementation [15]. The ultimate goal of our work is to create full system implementations (including both hardware and software) with security properties that can be directly measured and verified.

In particular, we are interested in verifying *information flow security* to capture security requirements such as confidentiality, integrity, and even real-time guarantees. The problem with verifying information flows through high-level specifications (hand-written or otherwise) is that these models often ignore predictors, caches, buffers, timing variations, and undocumented/unspecified instruction behaviors that are not part of the ISA-level specification of the architecture. These structures greatly complicate accurate information flow evaluations because they can be used to infer the values of secret keys [8, 19], to subvert documented protection mechanisms [28], and to covertly transmit information between compartments [23].

Recent work has shown how information flows can be *strictly* managed at the architecture level in a way that unifies explicit, timing, and storage information flows [30, 31]. This allows many of the security properties of the resulting

hardware/software system to be clearly and unambiguously analyzed. However, the current incarnations of these approaches have thus far come at the cost of drastically reduced flexibility. For example, an Execution Lease [30] allows for a region of execution to be tightly quarantined both in time and space (which can then be verified with gate-level information flow tracking), but requires that each and every lease have a *hard* time bound. After the bound expires, control is revoked from the leasee, meaning unbounded or timing variable operations do not fit cleanly into the model.

These restrictions present numerous challenges with regards to the creation of real systems. The hard nature of an Execution Lease programming model does not naturally support performance enhancing micro-architectural features such as caches, pipelining, branch predictors, or TLBs because of the timing variabilities they introduce; it lacks sufficient support for software behaviors that cannot be bounded and divided into fixed regular sized chunks of work, it does not provide any easily verifiable mechanisms by which different trust domains can communicate safely; and it makes handling the inherently dynamic nature of I/O very difficult. Furthermore, information flow security is provided using additional *analysis logic* that adds substantial area-delay overheads to the deployed system. In this paper, we present our experiences building a full system that removes all of the above restrictions yet is still verifiably information flow secure, i.e. conforms to a specified information flow policy.

Our method for overcoming this challenge is two-fold: first, we create a thin skeleton of hardware, that when configured and operated by a small piece of software, describes a *minimal* functionality with which the information flow of the rest of the machine can be governed. **In essence, this minimal slice of the hardware is to the entire processor core, as a microkernel is to a full operating system**. This strict structure is then enhanced by a runtime system that delivers more dynamic or non-information-flow-critical services (e.g. communication interfaces and context swapping). Because these operations are decoupled from the information flow properties of the skeleton, they do not add complexity to the verification of the system as a whole.

To make this idea more concrete, consider the process by which a context is saved and restored. To be a correct context switch, the registers and PC (along with other things) need to be saved off to a region of memory, and then restored when the process is scheduled again. However, to verify the *information flow properties* of such a system, we need only to verify that the context is saved and restored in a way that does not leak data and violate policy.

To demonstrate these principles we have created a synthesizable full-system prototype, complete with a pipelined CPU, a micro-kernel that enables isolation and communication by explicitly controlling all micro-architectural state, and an I/O subsystem that allows off-the-shelf I^2C devices to be connected to a single shared bus. Our system can provide caches, pipelines, and support for the micro-kernel in only 1/4th the area and with double the clock frequency as more restrictive prior work. Finally, for a system of size 50K logic gates (approximately) and with only 3264b out of 133kB state specified concretely, we can statically verify that the entire hardware-software stack conforms to a specified information flow policy all the way down to its gate level implementation.

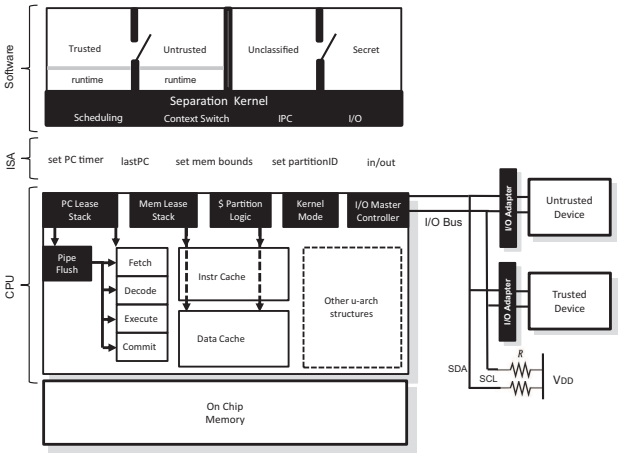

Figure 1: The proposed architectural skeleton (shaded black in the CPU) that allows explicit software control over the entire processor state. The processor includes dynamic micro-architectural features such as caches and pipelining. This hardware skeleton is used by a separation kernel to manage execution time, memory, and I/O devices among multiple security partitions. We also introduce trusted adapters for secure I/O. Here, an I^2C master controller on the CPU manages a shared bus among off-the-shelf I^2C devices with different trust levels. In the end, we verify that the hardware and kernel together enforce a desired information flow policy such as non-interference.

2. INFORMATION FLOWS AND RELATED WORK

Systems required to provide critical services must be engineered and evaluated to a very high level of assurance. For example, bus interfaces on aircraft multiplex critical aircraft control data with data of peripheral importance (such as passenger internet) on the same shared physical bus, and must guarantee that the integrity of the critical data is preserved under all circumstances [13]. Similarly, a program that uses the private key of a bank should be demonstrably confidential from other unclassified programs. While strict adherence to information flow policies alone is not sufficient to ensure the trustworthiness of a system, information flow policies are certainly necessary. For example, in the case of cryptographic operations, while there is clearly information flowing from the key to the encrypted data, we also need to ensure that the key is not flowing anywhere else (e.g., through a timing channel).

At the highest level there are two classes of approach to this problem: *best-effort* and *strict*. We define a best-effort approach as one that attempts to manage these information flows by closing known existing holes, managing uncloseable channels through statistical techniques such as clock fuzzing, and structuring hardware and software as a whole to make the job of the adversary as difficult as possible. While this is more than sufficient in many scenarios, the most that a best-effort approach can hope to achieve is a demonstration that, subject to the threat model, no known attacks are feasible. A strict approach, in contrast, carries a higher burden for proof – it should be able to show that, subject to the threat

model, no attack (known or unknown) is possible. The point of this paper is to demonstrate that, while this added burden is still great, it need not preclude many of the performance and productivity enhancing features we take for granted in more dynamic systems.

Information Flow Policies: In this work we target security properties such as confidentiality and integrity (where real-time performance guarantees are part of timing-integrity). Both these properties can be modeled with an information flow control lattice ($\mathcal{L}$, $\sqsubseteq$), where $\mathcal{L}$ is the set of security labels and $\sqsubseteq$ is the partial order indicating relative secrecy or integrity of the labels [12]. A simple lattice to represent integrity is Trusted $\sqsubset$ Untrusted; its dual lattice for confidentiality is Unclassified $\sqsubset$ Secret. Information flow policies are then specified as a mapping from security labels to each input and output in the system, and our goal is to verify whether it is possible under any scenario for this policy to be violated (e.g., untrusted data never flows out of a trusted output).

Expressing information flow policies using a label lattice has a few noteworthy characteristics. The policy of non-interference [14] allows for information to flow up in the security lattice, i.e. trusted data can affect untrusted outputs (for integrity) and unclassified information can flow to secret outputs (for confidentiality), and thus is represented by a totally ordered lattice. Complete isolation, on the other hand, is represented by a lattice comprising of mutually unordered labels. Further, in a real system, expressing security as a lattice of labels requires that the kernel have the lowest label, e.g. trusted or unclassified. Thus for integrity, the kernel parameters such as the schedule of security partitions or the partition boundaries themselves are trusted and cannot be tampered by untrusted programs. For confidentiality, kernel parameters are not secret and learning these values is not a security violation. To ensure security, the kernel has to ensure that secrets or untrusted values (i.e. high labels in either lattice) do not leak to outputs (including memory addresses) that have low labels.

Throughout this paper we use trusted and untrusted labels to enforce a policy of non-interference. However, the discussion applies to labels for secrecy and to labels in a general lattice.

Threat Model: We treat the entire system, including the processor hardware, micro-kernel, I/O controllers, and all the user-level programs, as a single monolithic unit for the purposes of verification. This unit has some known state bits (that are initialized with code and data for the micro-kernel and I/O master controllers), some unknown state bits (such as unspecified software components or unknown intital conditions), and a set of external input and output ports. The input and output ports are assumed to have a set of security labels, and an information flow policy is a specified as a lattice over those labels.

The attacker is assumed to have *complete control* over 1) all untrusted inputs to the device and 2) the entire set of bits which are unknown at analysis time. With our technique, we can statically verify that the type of data flowing to any output port conforms to the information flow policy for the label of that port (e.g. no untrusted information contaminating a trusted port, and no secret information leaking to a non-secret port). This threat model includes timing channels, implicit flows, storage channels, and any other digital forms of information flow, but does not include the use of physical phenomena such as EM emission or power draw. A system is said to be strictly enforcing a policy if it can be shown that the policy can *never* be violated regardless of the actions of the attacker subject to this model.

Formal Methods for Information Flow Security: While information flow tracking has been proposed at many levels of the computing hierarchy, from virtual machines [16], high-level languages [26] and compilers [34] to binary analysis tools [5] and even hardware-assisted information flow tracking systems [29, 11, 25], formal approaches for the same have been confined to language-level and operating system-level proposals. Approaches that operate at the language level can even track implicit flows due to branches and loops that introduce non-determinism into a program execution. Since code that is never executed can leak information (by the absence of its execution), some secure languages eliminate non-deterministic behavior from the program code (either entirely or based on confidential or untrusted conditionals [26]). At the OS level, Flume [22] has even been shown to be information flow secure through abstractions such as processes, pipes, file systems etc, while seL4 [18] uses automated theorem proving to verify safety properties of the kernel.

None of the above approaches account for covert flows through architectural state hidden beneath the hardware-software interface (i.e., the processor's instruction-set architecture), and timing channels created through shared architectural resources like on-chip buses, caches, branch predictors, and even functional units. Such covert channels have received isolated attention, for example methods to build secure caches [33] or branch predictors [6], but general design methods and tools are lacking. The techniques we propose are complementary to higher level approaches and would likely work best when informed by the rich semantic information available at the language level.

Information Flow Security at the Gate Level: While the approaches discussed above are very valuable, discovering many of the most subtle and exploitable information channels requires that we analyze designs at the level of logic gates, where the behavior and timing of the machine is actually well defined. Analyzing higher-level specifications cannot provide a strong guarantee that the full system *implementation* adheres to desired access control and information flow policies. For this reason, we build upon recent work on Gate-Level Information Flow Tracking (GLIFT) [31]. Based on the observation that *all* digital information flows are a function of *logical information flows* through the gates and wires in a circuit, GLIFT shows that a gate-level description of a processor can be automatically augmented with shadow logic-gates that dynamically track the flow of information through the processor and can identify information leaks through explicit, implicit, and even timing channels. Execution Leases [30] is an architecture that builds upon GLIFT and allows programmers to construct a stack of nested, space-time sandboxes. This stack of memory and execution-time bounds can be shown to be information leak free at the gate level, and having the benefit of knowing the exact signal values at run-time, can provide functionality such as secure pre-emption of untrusted programs based on trusted interrupts at arbitrary times.

One downside of a dynamic approach is that shadowing every gate in the circuit adds a considerable hardware overhead over the base logic (up to 3X [30]). Another significant downside is that dynamic techniques can only identify information leaks on specific executions and require the entire

system to be fully specified. High assurance systems often require static guarantees that a hardware/software system conforms to a specific information flow policy even when certain components are neither known a priori nor are trustworthy. In contrast to dynamic tracking methods, our technique allows properties of gate-level descriptions of systems to be verified statically even when large parts of the system such as user-level programs and peripheral devices are not known. Finally, the Execution Lease CPU provides a very restrictive programming model and implementation that does not support full systems efficiently. This paper addresses the above shortcomings by constructing a full system comparable to extant high assurance systems, a tool for static gate-level information flow analysis, and using the tool to verify a specific instance of the system.

3. A SECURE ARCHITECTURAL SKELETON

Our architectural skeleton, working in conjunction with the micro-kernel, must deliver each of the following capabilities in a way that can be verified to be side-channel free.

1. **Verifiable Common Case Optimizations in the CPU:** Techniques such as caches and pipelining are taken for granted in the non high-assurance systems but pipeline stalls and cache evictions can easily introduce side-channels. While countermeasures for these side-channels exist, we must be able to formally prove their absence.

2. **Verifiable Context Switches, Scheduling, and Communication in the kernel:** Because timing channels are part of our threat model, the micro-kernel, working in conjunction with the skeleton, must have a way to bound the behavior of a software partition. Furthermore, it must have a way to save and restore process contexts without leaking information about those contexts, to schedule these processes or partitions at arbitrarily fine granularities, and to allow inter-partition communication in a tightly-controlled manner.

3. **Verifiable I/O:** We must be able to construct a system that is able to communicate with the outside world. In particular we must allow software partitions measured access to I/O, and this access must be able to exploit simple off-the-shelf I/O protocols. While both authentication and physical attacks are beyond the scope of this paper, we must ensure that, if necessary, information flow can be limited to a subset of the parties connected on the I/O network.

4. **Verification Technique:** While prior approaches instantiate expensive information flow tracking logic, we aim to verify the concrete hardware implementation and the partial software specification together to be completely free of undesirable information flows. To this end, we propose a static analysis technique that represents all unknown values with the abstract value $*$ and replaces all hardware with logic capable of operating directly on $*$. This technique, $*$-logic (*Star-Logic*), then computes on $*$ and security labels to effectively verify information flows through all possible executions arising from the unknown values.

Understanding our approach to the problem can be difficult at first because our design method and verification

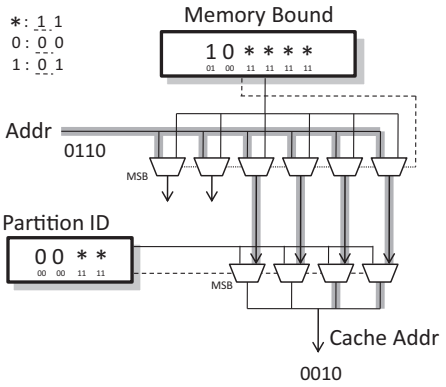

Figure 2: Implementing caches: The processor-generated memory address is first masked off used the memory bounds register. This creates a physical address that is within the currently active execution lease bounds. Most significant bits of this address are then further masked off using the trusted partition ID register to generate the address for the cache. As a result, information flow from a potentially untrusted memory access is limited to the currently enabled portion of the cache. Since trusted bits from the partition ID register control the MUXes, information flow control can be verified precisely.

method are intimately linked. **The verification method works because our designs exhibit incredibly tight control over the flow of information, and our design method is useful because designs can be verified easily.** Later, in Section 4, we will describe how we verified a specific incarnation of our system, but for now we can think at a high level about the two primary methods of managing the information flow in our skeleton.

The first method for information flow control is to ensure that certain critical portions of the machine and kernel are always kept with high integrity, i.e. trusted. If we can verify that the system will never breach this invariant, then these critical bits can be used as the root of trust for the rest of system.

The second method is carefully time-multiplexing the rest of the state between multiple security levels. There are two parts to this second method. Because these time multiplexed bits, for example the hardware's program counter or the kernel's current process ID, will change labels over time, we must bound the effect that these bits can have on the system. Then, after we have finished a unit of time, we must always be certain to "clean-up" any of these bits remaining in multiplexed parts of the system (as controlled by trusted bits).

In short, our minimal skeleton working with the separation kernel ensures that "trusted bits stay trusted" and that "untrusted bits always get cleaned up". Both these methods can then be verified to be implemented correctly through a gate-level analysis. Cleaning up untrusted bits in a verifiable manner is best understood using a multiplexer (MUX): if the select input to a MUX is trusted, and it selects a trusted value, the result can be trusted no matter what that actual value is, even when the other input is untrusted. A MUX can thus be thought of as a gatekeeper for trust, and is

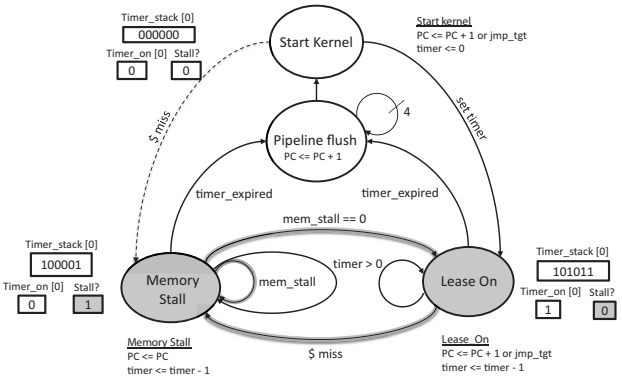

Figure 3: Secure Pipelines: The state machine shows the Program Counter update logic for a pipelined CPU. From all untrusted states (in gray), there is a transition to a trusted state that is triggered by a trusted timer, and hence the PC is always reset verifiably. The pipeline flush requires 4 cycles since our prototype CPU has a 4-stage pipeline. The dashed line from start to memory stall is to indicate that while a memory stall is possible, kernel code ensures that no memory access misses in the cache.

used to implement logic that resets critical system state to a trusted value or masks out untrusted signals. Together, both these methods allow, for example, the kernel bits that store the partition schedule to always stay trusted and control the MUXes to reset the program counter from an untrusted state.

In the rest of this section we will show how the combination of bit-level isolation and trusted time-multiplexing allows us to first implement a skeleton that addresses the four requirements listed above and then formally verify that the entire system conforms to a desired information-flow policy.

3.1 CPU: Using Caches, Pipelines, and Other Micro-architectural Structures

Caches. Side-channels through caches have been shown to leak information about private keys [7, 32]. These attacks exploit the ability of an attacker to learn the memory accesses of a secret process by first filling the cache, yielding to the secret process, and then inspecting which of its own memory accesses miss in the cache. The fundamental problem is that the cache controller uses both secret and unclassified information to decide which cache lines to evict. Counter-measures to this attack include pinning secret lines in the cache [32] (which was first shown to be vulnerable [20] and then fixed [21]), and proposed secure caches may still be vulnerable to collision-based timing-driven attacks [20].

To implement an information flow secure cache, we design the cache controller to only use trusted values to control the cache contents: i.e. by *partitioning* the cache or by *clearing* the cache based on trusted parameters after untrusted or secret code has finished executing. This ensures that untrusted information is confined to its partition, while trusted information can flow to untrusted partitions (or unclassified to secret).

To implement partitioned caches that are verifiably isolated at the bit-level, we allow only power-of-2 aligned cache partitions that are configured by the kernel through a `partition ID` register (as shown in Figure 2). The partition ID register represents currently enabled cache partition(s) and uses two bits for each bit of the cache address. Of these two bits, if the MSB is 1 the cache address uses the processor-generated memory address else the LSB of the partition ID register is used. For example, for a 4b cache address, a partition ID of 00_00_11_11 implies that the two most significant bits of the cache address will be 00, and the two lower bits will be used from the address generated by the processor. The partition ID register set to all 1s will indicate that the entire cache is available for use. The kernel sets the partition ID register before jumping to untrusted code using the instruction `set_partitionID immediate`. The cache controller also communicates with a memory controller in case of a read-miss or write-evict, and squashes an outstanding memory request when the execution time slot for some untrusted code ends.

This cache controller logic can be verified to be secure at the bit level because the MUXes that select the final cache address are controlled by the trusted partition ID register (Figure 2). While prior work has proposed that unclassified code pre-load AES tables into locked cache lines or clear the entire cache after secret execution [21], we are able to implement the mechanism and verify it at the gate-level.

Pipelining. Pipelines are challenging to implement in an information flow secure manner since they introduce unpredictable dynamic behavior through memory stalls, branch prediction, register forwarding etc. Our key insight is that such dynamic behavior can be allowed as long as untrusted programs' effects do not spill over into trusted space and time slots.

Figure 3 shows the state machine that controls the program counter (PC) in our CPU. The state machine begins in a `start` state where no PC lease is currently on and trusted kernel code is expected to execute. It can set a timer and transition to the `lease_on` state. This state is typically used to run untrusted or secret programs, but the transition to this state is based on a trusted jump instruction. On a cache miss, the state machine can transition to `memory_stall` state. From an untrusted instruction, this transition will be untrusted and the `memory stall` state will be untrusted too. When the PC timer expires, however, both `lease_on` and `memory_stall` states transition to a sequence of 4 states, one for each stage of the pipeline, where the PC is first restored to a trusted value (and then incremented each cycle). The logic to implement this sequence is hardwired instead of using the jump instruction because general purpose registers may themselves be untrusted at the end of a lease to untrusted code. Since the restore PC is stored in the trusted lease unit, computing the next PC from this maintains the PC as trusted. Further, since the lease timer has expired, no instructions are committed during this sequence of states.

Other micro-architectural features can also be employed safely through a combination of trusted partitioning and time-multiplexing. While we do not implement a branch predictor or prefetcher, implementing these would require their state to be either partitioned using trusted parameters or flushed at the end of every security context by the kernel.

3.2 Micro-Kernel: Context Switches, Scheduling and Communication

A kernel *partition* encapsulates all computational resources required by a security level, comprising of time, memory, and

optionally I/O interfaces. A portion of instruction and data memory are reserved for each security level, and when a partition is actually scheduled to run, it gains control over part of the machine such as execution units and register files for a trusted amount of time. To prevent information leaks to untrusted programs, kernel parameters such as time and memory slots allocated to each partition and the overall number of partitions depend only on trusted constants assigned at boot time. As a result, the kernel scheduler implements a statically determined schedule which can act as a coarse-grain first level scheduler, while each partition implements a second-level scheduler to optimize performance within their own time bounds (as proposed before for real-time [24] and highly secure [18] systems).

Precise Context Switches: The kernel ensures continuous unbounded operation by saving and restoring user programs' state on every context-switch. To support precise control over timing in the presence of caches and pipelines, we introduce two unique features in our micro-kernel. The kernel explicitly manages all micro-architectural state in the processor, e.g. through the partition ID register to enforce cache partitions, and has perfectly imperturbable execution time for each kernel function, e.g. by never having a memory access miss in the cache.

To demonstrate these ideas, we step through the context switch routine that is triggered each time a set timer expires. After a pipeline-length delay to flush the entire pipeline state, the first kernel instruction to commit is a `set_partitionID immediate`. This activates the kernel partition which stores complete context information for all partitions. The kernel then stores general purpose registers in trusted addresses specifically earmarked for the partition's context. This is possible since the number of partitions is a trusted kernel parameter. The kernel also stores the last PC that entered the commit stage when the timer expired, accessing it through the (`last_PC Mem[reg]`) instruction. Once the current context is saved, the kernel loads the general purpose registers for the new partition. It then sets the cache partition available for the next partition (using `set_partition`), sets memory bounds (`set_membounds global/local` and finally sets a time limit for the new partition to execute (`set_timer`). The `mode` argument to `set_timer` indicates whether the new context will execute in kernel or user mode (since the partitionID register can only be set in kernel mode). Finally, once the new partition's context is loaded and the bounds are set, the kernel loads the user-space PC and jumps to it.

Since the kernel has its own reserved partition in the cache, each of its memory accesses is a cache hit, and because there are no branches in its code, the kernel never has a pipeline stall. Thus context switches always complete within a fixed execution time. Further, the kernel explicitly controls the state of all micro-architectural features, using partitionID register for the cache, and relying on the processor skeleton to reset the state of the pipeline, memory controller and the I/O bus-controller when a lease timer expires. Finally, while not implemented in our prototype, the kernel can optionally save and restore the entries on the lease stack. Saving the lease stack for each partition allows schedulers that run inside partitions to use different scheduling granularities than that of the kernel, without being aware of the time bounds that they themselves run within.

Fine-grained Scheduling. We propose a novel hardware stack implementation that allows execution time and memory bounds to have arbitrary durations and yet is verifiably secure at the bit level. The time and space bounds for each partition are stored in a hardware stack in the CPU skeleton. This stack has to verifiably ensure that code inside a partition cannot over-write its own lease bounds, i.e. the current stack pointer should never affect any stack entry below the top of stack. The implementation in Execution Leases [30] protects lower stack entries by bit-encoding the timer values (e.g. each bit represents 32 cycles), and constructing each timer entry to have less time units than the previous entry. This restricts scheduling granularities to be aligned with the bit-encoding and introduces artificial performance overheads.

We provide a more flexible stack implementation by encoding the safety property in the logic for the *stack pointer* instead of the timer values (Figure 4). By choosing the stack pointer as the critical state, we free up the timers themselves to be assigned arbitrary values. We encode the stack pointer so that each bit corresponds to a stack entry, and on every clock cycle, the value assigned to each bit of the stack pointer is predicated upon *all* the lower stack pointer bits being true; i.e. a stack entry is valid only if all timers lower than itself are still active. Thus while information clearly flows from lower stack pointer bits to higher ones, higher bits never affect the lower ones. Our two information flow control techniques can be observed here: while timer[0] is permanently trusted, the rest of the stack is time-multiplexed between both security levels.

We modified the `set_timer` instruction to receive arbitrary values as the time limit, and added a `mode` bit that a caller can use to specify whether the callee executes in kernel or user mode. This bit is set to 0 by the kernel when it schedules a user-space code in order to protect the partition_ID register. Our CPU implementation has three separate lease stacks of 2 entries each, one for execution time (PC) and one each for local and global memory bounds.

Read/Write Protection: Information flow policies such as non-interference [14], Biba [10], and Bell & La Padula [9] allow information to flow along one direction in a security lattice, e.g trusted parameters can be read but not tampered with. In order to support security policies that allow unidirectional information flow, we modify the lease implementation to support read-only, write-only, or read-write control over memory bounds. In contrast, the original implementation of Leases allowed complete access to memory regions that are specified as part of the lease and could only support isolation in memory. The kernel uses the instruction `set-membound global/local, timer, addr_range, RO/WO/RW` to specify whether the memory region is Read-Only, Write-Only, or Read-Write, and the memory controller enforces these permissions at run-time.

To enforce isolation, we specify non-overlapping memory bounds for the partitions[1]. To implement communication, one option is for two communicating partitions to share a memory region. This requires the partitions to be initialized so that communicating partitions are adjacent. Since our CPU implements two distinct memory regions per partition, one partition can share memory areas with a maximum of four other partitions to facilitate zero-copy communication.

[1]Embedded systems typically operate exclusively on physical memory, eliminating channels associated with both aliasing and dynamic allocation

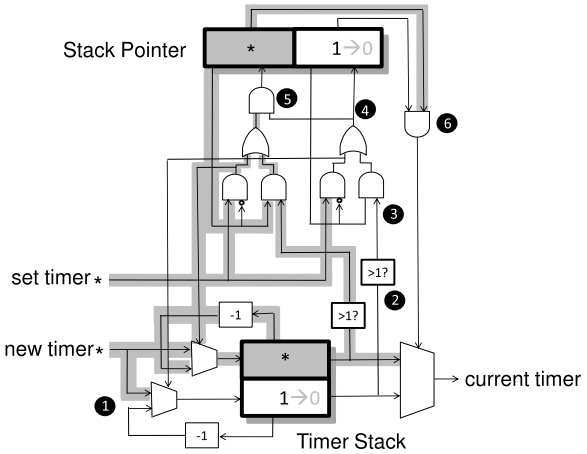

Figure 4: Implementing flexible timers with bit-level isolation. Isolation: Untrusted code (in gray) does not taint the lowest stack entry (1), and when the timer expires, the `current timer` value becomes trusted (path 2-3-4-6). A timer, when it expires, resets all timers above itself (5) and thus enforces nested leases. Flexibility: Instead of timer values, the `stack pointer` is encoded to ensure bit-level isolation. Hence the actual timers can be assigned arbitrary values.

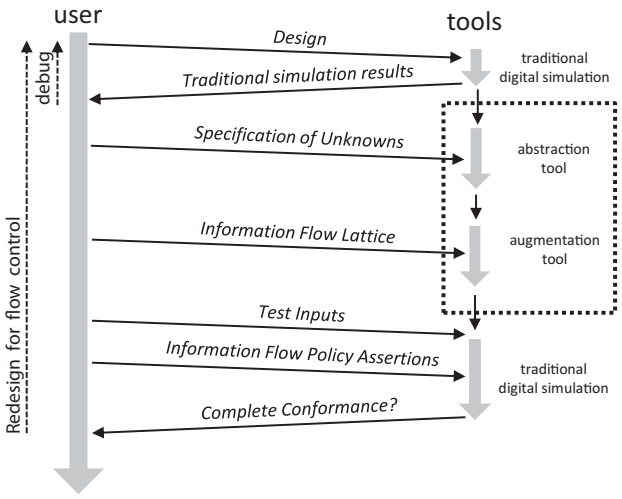

Figure 5: Our toolchain for verifying information flow properties of embedded systems. Once the design has been debugged using conventional tools, our abstraction and augmentation tools create a new design that operates on security labels and unknown values in addition to traditional digital values. This augmented design can then be simulated using standard hardware simulators to generate output labels. These are then compared with labels specified by the desired information flow policy to determine if the design conforms to the policy.

For more complex communication patterns, the microkernel has to move data into partition "inboxes".

3.3 I/O: Using Off-The-Shelf Protocols and Devices Securely

We complete our embedded system by implementing an I/O protocol to connect the CPU and separation kernel system to peripheral devices. I^2C is a serial two-wire bus protocol that is commonly used in many embedded systems, for e.g. to configure RF tuners, video decoders and encoders and audio processors. It is also present on chipsets designed by Philips, National Semiconductor, Xicor, Siemens, and many others [17]. We show, for the first time, that it is possible to implement a provably secure I^2C master controller that can then interface with commodity I^2C devices.

Implicit information leaks occur when, for example, the I^2C bus master first communicates with an untrusted slave, and then with a trusted slave. The master's current state will first depend on information received from the untrusted slave, and appear as a covert channel underneath the ISA to software, where the untrusted slave affects the timing of trusted communication. At the gate-level, this implicit flow takes the form of an explicit ACK message from the untrusted slave to the trusted master's state machine, causing the master's state machine to be labeled untrusted. Thus even if the bus master seems to be behaving "correctly" and the devices are not snooping on the bus, there are still information flows between devices on the bus.

Trusted Bus Adapter: To restrict these implicit flows, we propose to overlay a time division multiplexed (TDMA) schedule over an I^2C bus, and introduce *adapters* to connect external devices to the shared system bus (Figure 1). Each adapter's time slot is a trusted kernel parameter, and the adapters enforce that for each slot only the currently addressed device has access to the bus while the remaining adapters disconnect their corresponding slave devices. When a lease timer expires, the bus master state machine is reset to a trusted state to eliminate the implicit flows mentioned earlier. We implement the adapters to not only impose a TDMA schedule on the bus but also conform to the I^2C specification. The adapters do so by driving the clock signal to a device low when its slot has expired, and since the I^2C protocol doesn't rely on wall-clock time, the devices hold on to data until the I^2C clock goes high again. As a result, we can use *unmodified I^2C-compliant devices* in our I/O subsystem.

The CPU - I/O Interface: The I^2C Bus Master state machine is implemented in hardware as part of the CPU. Programs in each partition (device drivers local to each partition) use two instructions `in dest_reg, dev_addr` and `out dev_addr, src_reg` to transfer a 32b value between a register and an I^2C device that the partition is allowed to address. Since peripheral devices are typically slower than a CPU, a device driver can use the instruction `i2c_on dest_reg` to record into `dest_reg` whether the I^2C bus is currently in the middle of a transmission.

We have described a system that manages the flow of unknown and untrusted bits such that arbitrary untrusted computation is tightly bounded by trusted bits. The challenge now is to verify that an implementation of the high-level description correctly enforces an information flow policy.

3.4 Verification: Removing Analysis Logic Overhead

Given a system in which the hardware and software cooperate to enforce a specific policy, we want to verify that the measures taken by a small, known portion of the soft-

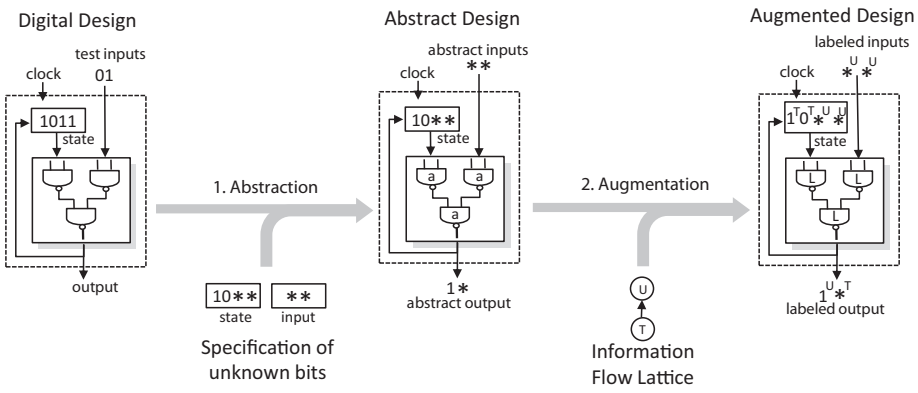

Figure 6: Given a concrete design to be verified and a specification of system state and inputs whose values are unknown, Step 1 of our technique automatically generates an *abstract representation* of the design that covers all possible concrete executions. In Step 2, the technique augments the abstract system with the state and logic to track the flow of security labels. The resulting *augmented system* is then synthesized and simulated using hardware design tools such as ModelSim and Altera Quartus. Augmented logic uses security labels specified by the user for inputs and memory bits and generates output labels that are compared with the desired output labels as specified by the information flow policy.

ware and hardware will be sufficient to prevent unauthorized labels from ever appearing at certain memory locations and output ports. This policy of *non-interference* prevents untrusted information from leaking to trusted ports, but allows trusted information to appear on untrusted output ports (or memory locations).

Our static analysis technique, $*$-logic (*star-logic*), works on a given hardware-software design in three steps: first it creates an *abstract* design that soundly estimates all possible executions of the given design, then it augments the abstract design with information flow tracking logic, and finally simulates this augmented design until all possible states of the abstract design are explored. Figure 5 shows how these three steps fit into the traditional hardware design flow, and Figure 6 presents the individual steps in more detail.

Step 1: Abstraction. The designer implements a system's trusted computing base and specifies the rest of the system as unknown (represented as $*$), where the unknown part represents the contents of memory for all the secret and unclassified processes and all the input ports. The $*$-logic tool takes the concrete implementation and a specification of unknown parts to generate an *abstract system* that operates on both known and unknown values. An abstract 2-input AND gate is thus a logic function from $\{0, 1, *\} \times \{0, 1, *\} \rightarrow \{0, 1, *\}$, and evaluates its inputs to decide whether the output is a 0, 1, or unknown. This tool builds upon the observation in GLIFT that if one input of a 2-input AND gate is a 0, the output is a 0 and the other input is ineffective even if it is unknown ($*$). For other input cases, the unknown input will cause the output to be unknown. The purpose of this abstract system is to track the flow of unknown values through the system, and in doing so strictly over-approximate all possible executions of the partially specified system. As a result, simulating the abstract design will mark as unknown all state that can ever be affected by the unknown inputs. This soundness property can be proven formally using the Abstract Interpretation framework [27], but the formal proof is out of scope of this manuscript.

Step 2: Augmentation. The augmentation step takes two inputs as well: the abstract design from the first step, and an information flow lattice (such as trusted $\sqsubset$ untrusted) that specifies a set of labels and implies rules for how they they are to be propagated. This second tool then convolves the lattice with the abstract design to create a new design that operates on both abstract values and labels, effectively propagating whether each bit is unknown and/or untrusted. Using the truth table for an AND gate, we can construct an augmented truth table where the inputs can assume a value that is one of $\{0^U, 1^T, 0^U, 0^T, *^U, *^T\}$ (encoded respectively as $\{000, 001, 010, 011, 110, 111\}$), and the output is computed to be one of the same. The first bit in the new tuple of values that the gate will operate on is 0 if the original value is concrete or 1 if the value is unknown. The second bit is the actual value if known and 1 otherwise. The final bit is 0 for U and 1 for T. Thus an n-input m-output digital gate is replaced by a gate with $3n$ inputs and produces $3m$ outputs. These augmented gates are then interconnected just as original gates of the system were. The augmented logic, again being a hardware design itself, can then be simulated using existing hardware synthesis and simulation tools.

Step 3: Simulation. The final step in the process is to simulate the resulting augmented system to check that it conforms to a specified information flow policy for every possible state of the augmented machine. Covering the set of all possible logic states, is made tractable by the abstraction from step 1, and because the concretely specified system that comprises the trusted computing base only has a practically enumerable number of states (a scenario very common to high assurance systems). To specify information flow policies a designer has to initialize the system by assigning security labels for each input, output, and memory location. During simulation, if an output or some state bit of the system is found to have an illegal label (e.g. a trusted output port has an untrusted label), then the system under test has to be modified for correct information flow control. In this way dataflow assertions (such as "no classified information should egress the system via port 2") can be checked as standard logic assertions ("the label bit for out-port 2 should never be set to True").

4. RESULTS

Figure 5 shows our toolchain to analyze hardware designs written in behavioral Verilog or VHDL (so that hardware designers can use their tools of choice for design entry). Verilog/VHDL designs are synthesized using Synopsys Design Compiler into a gate-level netlist using the `and_or.db` library. The result of this synthesis is a netlist that consists of just AND, OR, and NOT gates along with registers and memory. This netlist is input to our abstraction tool, which replaces gates and bits of the netlist with their abstract counterparts and outputs the abstract netlist. The abstract netlist is then input to our augmentation tool that generates information flow tracking logic for the abstract design to create the final netlist. Finally, the augmented design is simulated using hardware synthesis and simulation tools such as Altera Quartus.

4.1 CPU Implementation

This section presents implementation details of our CPU (*Star-CPU*) and compares its functionality and area-delay with prior work. We implemented the Star-CPU in Verilog, generated a gate-level netlist using Synopsys Design Compiler, and synthesized this using QuartusII v9.1 with Altera EP2S15F48C43 FPGA as the target device. The Star-CPU pipeline is single-issue, executes in-order, and has 4 stages (fetch, decode, execute, and commit/write-back). It has 8 general purpose registers, a mode bit to indicate kernel/user mode, and a partition ID register to record the current security context. The memory hierarchy includes a 2kB direct-mapped data cache, and 64kB each of instruction and data memory. The data cache is implemented on the FPGA using comparator logic and registers and requires one cycle if a memory access is a hit, while the memory is implemented using on-chip block RAMs that take two cycles to service a memory request. To emulate memory access latency in an ASIC implementation of the system, the memory controller is implemented to introduce an additional delay of 100 cycles. Without micro-architectural features such as branch predictors, TLBs, and Out-of-Order execution, the Star-CPU pipeline stalls on each cache miss and requires the compiler to ensure that a register used in a conditional jump instruction has the desired value at least 4 instructions before it is used.

Area-Delay Comparison. Figure 8 quantifies the size and performance advantages of the Star-CPU against the Execution Lease CPU and against the Star-CPU with dynamic GLIFT logic (Star-GLIFT). The Star-CPU provides caches, pipelining, and kernel support beyond the Lease CPU in equivalent area and clock-frequency, and provides static security guarantees compared to Star-GLIFT in almost 1/4 the logic, 1/2 the memory, and 2X the clock-frequency.

The Star-CPU's base functionality is implemented in 5756 ALUTs (Adaptive Look-Up Tables in an Altera FPGA, where 1 ALUT corresponds very approximately to 9-12 gates), and while the base functionality in the Lease CPU requires only 1511 ALUTs, it requires 5040 ALUTs when the dynamic analysis logic is factored in [30]. Thus the Star-CPU replaces analysis logic overhead with a cache and pipeline logic. In terms of performance, the Star-CPU and Lease CPU have similar frequencies (99 MHz vs. 104MHz), but the unpipelined Lease CPU only commits one instruction every 5 cycles. Further, without a cache, every memory access in the Lease CPU goes to main memory.

Instruction	Description
set_timer R1, R2, R3	Set PC lease. Arguments R#: register or immediate. R1: timer, R2: restore PC, R3: kernel/user mode
set_membound R1, R2, R3	Set local or global memory bounds. R1: memory range, R2: timer, R3: read/write mode
set_partitionID Immediate	If mode == kernel, then partitionID = Immediate
last_PC [R1]	Mem[R1] = PC in commit stage when the last timer expired
jgtz /jump R2, R1	Jump if R1 >= 0 or unconditionally. PC = R2 or Memory[R2]
load/store/mov R2, R1	Immediate and register direct addressing modes
add,sub,lsh,rsh and,or,not,cmplt R1, R2, R3	ALU instructions. Register arguments
in/out R1, dev_addr	Read and write to I^2C transfer register
io_on R1	R1 = 1 if I^2C transaction is ongoing
no-op	No-op instruction

Figure 7: Figure shows the ISA for the Star-CPU

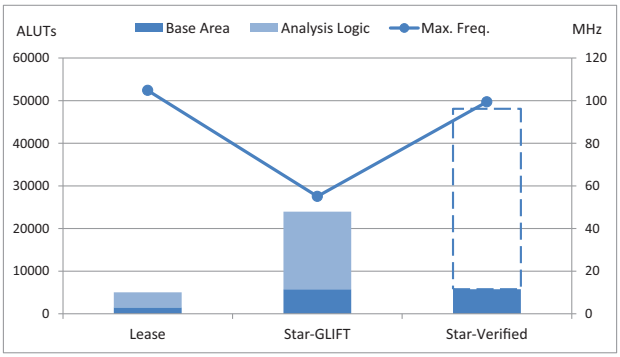

Figure 8: Area and Frequency comparison among secure CPUs.

Comparing the verified Star-CPU to Star-GLIFT, we observe that the Star-GLIFT CPU requires 23,956 ALUTs for logic and 2×133kB for state and state labels, whereas the Star-CPU only requires 5756 ALUTs and 133kB for state. Adding dynamic tracking logic for the complex control logic of the CPU introduces substantial delays and reduces the maximum operating frequency of the Star-GLIFT CPU to 55MHz (from 99MHz for the verified Star-CPU). In summary, the verified Star-CPU provides better functionality than the Lease CPU, and static verification in comparison to Star-GLIFT CPU with much lesser area and delay.

4.2 Kernel Implementation

Our full-system prototype is representative of a high assurance avionics system [1]. Each trust domain in the system is assigned a partition, and a micro-kernel manages these partitions' access to the CPU, physical memory, and peripheral devices. The kernel implements an ARINC 653 scheduler (a standard in avionics systems) which requires that all partitions be statically defined at compile time in the form of a *major frame period* that repeats forever. Within a major frame, the schedule specifies one or more execution time slots for each partition, leaving the partitions free to implement standard, priority-based schedulers within their time slots.

Specifically, our prototype instantiates 4 partitions: the first partition to run programs responsible for controlling trusted avionics functions, the second partition for untrusted programs such as passenger internet and non-critical diagnostics, the third partition for a trusted cross-domain guard responsible for one-way communication among the above two partitions, and the fourth partition reserved for trusted kernel functions that require hardware access such as a con-

```
Context Switch 1: // from partition 0 to 1
    // save current state: 10 cycles
    set_partitionID   KERNEL_ID;
    store  ctxt_arr [PID0][0],  gen_reg [0]; …
    store  ctxt_arr [PID0][7],  gen_reg [7];
    last_PC  ctxt_arr [PID0][8];

    // restore incoming state: 9 cycles
    load  gen_reg [0],  ctxt_arr [PID1][0];  …
    load  gen_reg [7],  ctxt_arr [PID1][7];
    set_partitionID   PID1;

Kernel Scheduler 1: // uses reserved registers R1 and R2
    // set bounds and timers for partition 1: 10 cycles
    load R1, PID1_BOUNDS_G; load R2, PID1_TIME_G;
    set memboundsg R1, R2, PID1_RW_G;

    load R1, PID1_BOUNDS_L; load R2, PID1_TIME_L;
    set memboundsl  R1, R2, PID1_RW_L;

    load R1, PID1_TIMER; load R2, Context Switch2_PC;
    settimer  R1, R2, USR_MODE;

    jump  ctxt_arr [PID1][8];
    // returns to context switch2 () in kernel mode

Context switch time = 2 x pipeline depth + context save + context restore  + kernel scheduler   = 37 cycles
```

Figure 9: Kernel scheduler and context switch functions in assembly. Security policies are expressed through the values of partition parameters for memory and time bounds. The functions are small since the ISA and CPU are designed for information flow control.

text switch. To effect a context switch, the kernel partition gets one time slot after each of the other partitions' slots. In actual systems, the partitions are sized so that they meet hard real-time guarantees demanded by critical application; we opt for arbitrary durations for each partition in order to demonstrate how to verify non-interference between the trusted and untrusted partitions.

To verify non-interference, we instantiate the trusted kernel scheduler and the context switch partition with concrete values, while the other three partitions are instantiated as unknown ($*$). The cross-domain guard partition overlaps a read-only memory region with the trusted avionics partition and a write-only partition with the untrusted partition. This ensures that information can only flow in one direction from trusted to untrusted. Note, however, that cross-domain guards can also be required to impose restrictions on the *type* of information that can be transferred. Enforcing such rules requires verifying the guard program logic using alternative formal verification techniques; bit-level information flow analysis is too coarse-grained to provide such guarantees.

The scheduler is small; the trusted, concretely specified scheduler and context switch code only requires **87** assembly instructions. This is primarily because the ISA in Figure 7 is explicitly designed for information flow control. The time required to switch contexts is an important performance metric for a kernel. In our system prototype, it takes the kernel partition **37 cycles** to switch one partition and schedule another: 4 cycles each to flush and re-fill the pipeline, 19 to save and restore partition ID, general purpose registers and the last executed PC, 10 to set new memory and time bounds and to jump to the PC for the restored context.

4.3 I/O Implementation

The I^2C devices and adapters are also processed using the verification flow mentioned above (note that the CPU and kernel can be verified independently of the I/O by treating the I/O interface in the CPU as trusted and unknown). Our experiment use a single master and three slaves connected to the bus using adapters. Each adapter synthesized individually requires 49b of state and the slave requires 21b. We wish to verify that information does not flow from the untrusted slave to any other device, and set up a test where the master is trusted and known, communicates with an untrusted slave that is unknown ($*^U$). The I^2C bus has two trusted slaves, one specified and one unknown.

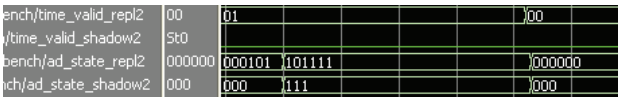

Figure 10: Figure shows how to check for a safe reset of the I^2C adapter to a trusted state (based on a trusted signal `time_valid`). The Modelsim simulation waveform begins with the adapter in an untrusted time slot (`time_valid = 1`). During the untrusted communication, the adapter's state bit stays unknown (as indicated by the MSB of `ad_state` being 1) and untrusted (`ad_state_shadow` = 1) until the time slot expires (indicated by the `time_valid` signal). At that point, the adapter's state machine is reset to a trusted state (indicated by the `ad_state_shadow` signal going low).

4.4 Verification Results

We simulate the augmented designs of the CPU and the I/O system for one loop of the kernel scheduler, come back to the initial state, and verify that all security labels for memory and outputs follow the desired policy for every state of the system. Figure 10 shows a screenshot of verifying that the I^2C adapter state is reset to trusted once an untrusted time slot has ended. We simulate a complete I^2C transaction, letting transmitted data values be unknown, and verify that no untrusted value ever appears on the adapter outputs to the trusted slave devices. This experiment used 3 slaves and 3 adapters with a total of 184 state bits, and if we assume that the hardware modules' contributions are proportional to their individual sizes, our technique can verify the 184b system by specifying ∼128b concretely and evaluating all combinations of the rest in a single execution of the augmented system.

The $*$-logic verification technique scales to handle large embedded system designs, as shown in Figure 8. Of the total 133kB state for the Star-CPU, only 3264b are required to specify the micro-kernel's scheduler, context switch code, and partition bounds. The verification scales because its complexity grows linearly with the size of the design under test, as each module is replaced by its augmented module.

Total verification time includes the time for both synthesizing the design and simulating it for a kernel scheduler loop. The augmented logic takes considerably longer to synthesize as compared to the basic design under test. The augmented Star-CPU, with 48093 ALUTs, required 14 hours to synthesize with QuartusII v9.1 as compared to just 7 minutes for the basic design with 5756 ALUTs. Once synthesized, simulating one loop of the kernel scheduler only takes a few seconds. All measurements were made on a 1GHz AMD Athlon 64 X2 Dual Core Processor with 1MB cache and 2.7GB RAM.

In the future, we will integrate $*$-logic with other formal techniques that can work with richer abstractions. While such techniques do not readily scale to large systems, these can complement $*$-logic to verify that the system is secure for a set of implementations or kernel parameter values instead of one specific implementation.

5. CONCLUSIONS

Embedded systems are trusted by people to do everything from stopping their cars to controlling the beating of their hearts, yet all too often these systems compromise strong

security for rich functionality. We have shown, for the first time, that complete statically verifiable information flow security is compatible with the convenience of continuous, unbounded operation and dynamic optimizations – even when we consider timing channels and other hardware/software leaks as part of our threat model. Our system is designed around an architectural skeleton that allows a micro-kernel to safely multiplex mixed-trust programs on the hardware, and can do so in $1/4^{th}$ the area and with double the clock frequency as more restrictive prior work. These advances are due in part to the development of a tool that can statically verify information flows through full systems at the bit level, allowing us to verify a 133kB system by specifying only 3264b concretely, and leaving behind the hardware dynamic flow tracking considered in prior work. While more work is required to examine the broad applicability and scalability of this approach, by implementing and verifying a full-system prototype (including a CPU, a micro-kernel, and I^2C based I/O) we have demonstrated that a useful balance between flexibility and hardware information leakage is not only possible but can even be relatively efficient.

Acknowledgments

The authors would like to thank the anonymous reviewers for insightful comments on this paper. This work was funded in part by Grant No. CCF-0448654, CNS-0524771, CCF-0702798, and the US Department of Defense under AFOSR MURI grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

6. REFERENCES

[1] Arinc 653. http://www.lynuxworks.com/solutions/milaero/arinc-653.php.
[2] Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/cc/.
[3] The integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.
[4] What does cc eal6+ mean? http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/.
[5] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.
[6] O. Accigmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *Cryptology, The Cryptographers Track at RSA*, pages 225–242. Springer-Verlag, 2007.
[7] O. Acicgmez. Yet another microarchitectural attack: Exploiting i-cache. In *14th ACM Conference on Computer and Communications Security (ACM CCS)*, 2007.
[8] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *The Cryptographers Track at RSA Conference*, pages 225–242, 2007.
[9] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical report, Technical Report MTR-2547, 1973.
[10] K. Biba. Integrity considerations for secure computer systems, 1977.
[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.
[12] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
[13] D. Federal Aviation Administration (FAA). Boeing model 787-8 airplane; systems and data networks security–isolation or protection from unauthorized passenger domain systems access. http://cryptome.info/faa010208.htm.
[14] J. A. Goguen and J. Meseguer. Security policies and security models. *Security and Privacy, IEEE Symposium on*, 0:11, 1982.
[15] D. Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, 2009.
[16] T. Jaeger, R. Sailer, and Y. Sreenivasan. Managing the risk of covert information flows in virtual machine systems. In *ACM Symposium on Access Control Models and Technologies*, France, June 2007.
[17] D. Kalinksy and R. Kalinsky. Introduction to I^2C. *Embedded Systems Programming*, 14(8), August 2001.
[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *SOSP '09: 22nd Symposium on Operating Systems Principles*, pages 207–220, NY, USA, 2009.
[19] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.
[20] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, CSAW '08, pages 25–34, New York, NY, USA, 2008. ACM.
[21] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 393 –404, 2009.
[22] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.
[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM conference on Computer and communications security*, pages 199–212, NY, USA, 2009.
[24] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.
[25] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 35–45, New York, NY, USA, 2008. ACM.
[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.
[27] D. Schmidt. Foundations of abstract interpretation.
[28] O. Sibert, P. A. Porras, and R. Lindell. An analysis of the intel 80x86 security architecture and implementations. *IEEE Transactions on Software Engineering*, 22(5):283–293, 1996.
[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.
[30] M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Intl. Symposium on Microarchitecture*, 2009.
[31] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.
[32] Z. Wang and R. Lee. New cache designs for thwarting cache-based side channel attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.
[33] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, 2007.
[34] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.